Flipped Session Short Answer – Week 2

Instructions: Work with a group (2-6 people) to answer as many questions as possible.

1. Give a concise description of the fundamental difference between a linked list of nodes and an array.
   A linked list can be resized at no cost, but does not support random access.

2. In the resizing array implementation of the stack, how does the line s[N] = null help reduce memory usage?
   Avoids loitering.

3. Is it a good idea to include a peek() method? Peek lets you see what is next on the stack without actually removing it.
   There's no reason not to. Simple to implement and does not complicate the API very much.

4. Arrays have functionality that Stacks do not utilize, in particular the ability to access items anywhere in the array in constant time (including the middle). Does this suggest that arrays are a poor choice since they aren't fully utilizing the power of the array? Why or why not?
   Nope. What actually matters are the tradeoffs with other possible implementations.

5. Does every operation take constant time in the worst case for the fixed-array-size implementation of stacks?
   Yes.

6. Is there any advantage to downsizing the ResizingArrayStack array other than saving memory?
   No.

7. Is the memory savings gained by shrinking the ResizingArrayStack array worth the cost in time?
   It depends on the application. In general, I'd say yes, since we're clearly already willing to pay linear time for some insertion operations, so most likely we are also willing to do this for deletes. If we really need guaranteed fast deletes and don't care about memory very much, we might avoid downsizing on deletes (but in this case, a linked list is probably a better case anyway since we don't care about memory).

8. One could argue that when you downsize an array, you're probably going to have to upsize it later. Why bother downsizing?
   It depends on the application. For a general purpose stack class that may be used by millions of people around the world, and instances of which may live for many years, it's probably not a good idea to have the property that the stack uses as much memory as it has EVER required.

9. What do you think about the idea of providing the user of your Stack class with the ability to manually specify when to increase/decrease the size of the array?
   This seems like a potentially high burden on the programmer, and may result in bugs in client code. However, it could be useful for fine tuning code performance.

10. A double ended queue is more flexible than both the stack and the queue and would use almost the exact same implementation. Why don't we just make all queues double ended (capable of adding and removing items from either end)?
    Ideally, we want to use the simplest possible abstractions when programming. As Devo noted in the 80s, mankind desires not freedom of choice, but freedom from choice.

11. Is the placement of the beginning and end arbitrary? Is there an advantage to doing things as in the lecture notes over the alternate: adding nodes to the start of the list and removing them from the end?

This question was a bit unclear, but it was talking about queues. The choice is entirely arbitrary (though I guess the way we do it in class matches how queues/lines work in real life).

12. When else might we use generics other than to implement stacks and queues?
    Any other collection, e.g. bags, priority queues, symbol tables, etc.

13. We saw that we could iterate through our Stack with code like:

    ```
    for (String s : stack)
      StdOut.println(s);
    ```

    Why would we ever use an Iterator<> instead of the code shown above? (I'm not sure I fully understood this question, but it was certainly popular on FlipYourTraining).
    Still never grokked this question.

14. What uses are there for the Bag data structure? Wouldn't it be simpler to just use a stack or queue and never pop?
    Again, we want to use the simplest possible abstraction. We will actually see bags in use when we get to graphs, but indeed one could simply not use pop/dequeue and performance and behavior would be exactly the same.

15. "Why is there Iterable and Iterator? Couldn't both be wrapped up in a single structure without losing any functionality, rather than having to retrieve an extra class and use that?"
    There IS just one structure – the iterator. Saying that you implement the "Iterable" class is a way of letting clients know that you provide that structure.

16. Our sorting algorithms take as input an array of Comparable[]. To be a Comparable, an object of type TypeX must simply implement the Comparable interface. One student asked ' couldn't you technically put anything into this array, even if it isn't comparable with the other elements, as long as it is from a class that implements Comparable?'. Why is this not an issue in Java?
    Java only allows an array to contain elements of a single type, thus all the comparables must be of the same type.

17. Would we be able to make a class that is comparable to other types of objects? For example, could we make a class similar to String that could also be compared to integers? How would we do this?
    Yes, though not with the comparable interface. We'd need to use a comparator (or a special purpose method).

18. "Does an interface do more than just list the functions that must be implemented? I understand that it's a very useful tool to have a consistent list of functions across objects, but would it ever cause an error to avoid implementing the interface if you would still implement all the necessary functions?"
    For 226 purposes, this is all an interface does. You can implement the functions without claiming that you implement the appropriate interface.

19. For shellsort, what is bad about the increment sequence 1, 2, 4, 8, 16, 32, …?
    Odd and even items are never compared until the last pass. Since N=1 is the same as insertion sort, this means we expect N^2 performance.

20. Insertion performs .25N^2 comparisons and .25N^2 exchanges is .5N^2 operations, which seems comparable to selection sort's .5N^2 comparisions and N (lower order and thus unimportant) exchanges. Yet Bob mentioned that we expected selection sort to be twice as fast based on the comparisons. Did Bob misspeak?

Though it might have seemed Bob as implying that comparisons were the whole story, they are not. I'm not certain if he misspoke. Kevin and Bob (and others) have made the empirical observation that insertion sort tends to be in the neighborhood of twice as fast as selection sort, though this varies with language, machine, and even the version of Java. To quote a recent email from Kevin:

"The 2x speedup was an empirical observation, at least in Java 1.5 (and I think in C and C++). I think we were seeing about a factor of 1.7x (see p. 256). But, surprisingly, when I just ran it, I see only a speedup of about 1.2x. If I run it in "interpreted mode only", then it goes back to around 1.8x. Also, if I compare InsertionX (optimized to do half-exchanges), it's about a 1.6x faster than selection sort. Not sure what the compiler is doing differently. My recollection is that, previously, InsertionX was not a big improvement over Insertion in Java. But it looks like the story has changed. Perhaps exchanges are now relatively more expensive.

I think the previous empirical observations could be explained by the compares dominating the running time - even though insertion sort is doing more array accesses, they're consecutive in memory, so caching makes them cheap."

21. Extra: Is it actually possible to implement precedence order with the Dijkstra two stack algorithm?
    In Polish notation, there is no notion of precedence order.